Projektbeskrivning

Lösenordshanterare

2021-05-27

Projektmedlemmar:
Edvin Schölin <edvsc779@student.liu.se>
Wilmer Segerstedt <wilse150@student.liu.se>

Handledare:
Jonathan Falk <jonfa001@student.liu.se>

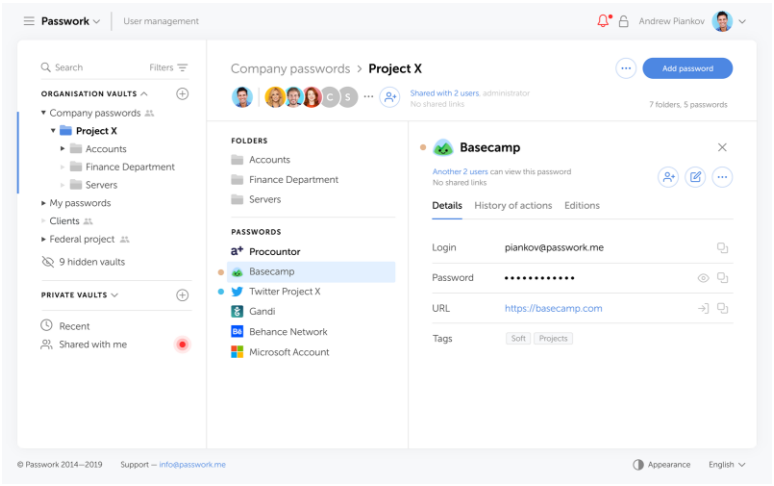
Contents

Projektplan

1. Introduktion till projektet

Vi vill utveckla en lösenordshanterare med funktionalitet att spara och kryptera inloggningsuppgifter. Allt användaren behöver komma ihåg är ett huvudlösenord för att kunna hämta önskat lösenord. Detta möjliggör att användaren kan använda unika och bättre lösenord för ökad säkerhet. Användaren kan interagera med programmet med hjälp av ett fönster där all funktionalitet kommer att finnas tillgänglig.

Tanken med figuren (figur 1) nedan är att ge en ungefärlig bild av vilken funktionalitet som kan finnas tillgänglig i vår lösenordhanterare. Lösenordshanteraren i det här projektet kommer inte ha samma snygga grafik och med säkerhet inte ha lika mycket funktionalitet, men figuren kan ge en bild av hur en lösenordshanterare *kan* se ut.



Figur 1. En bild på en lösenordshanterare.[1]

2. Ytterligare bakgrundsinformation

Kryptering är en metod för att göra information svårläslig för någon person som informationen inte ska vara tillgänglig för.[2] Detta är kritiskt för vår lösenordshanterare där bara användaren ska kunna få tag på information om sina inloggningsuppgifter. För att kunna få tag på lösenorden krävs en dekryptering av dessa.

I detta projekt vill vi använda Java Cryptography Extension (JCE) för att kunna kryptera och dekryptera lösenorden. JCE gör det möjligt att peka på ett värde (lösenordet) utan att veta det själva värdet.

---

[1] Bild på *Passworks* lösenordshanterare.
Hämtad från: https://passwork.me/pages/cloud/images/en/hero1@2x.png, 21-04-29.

[2] Wikipedia. *Kryptering*. Kryptering – Wikipedia, 21-04-29.

Algoritmen vi valt för att kryptera lösenordet kallas för AES-256. Denna algoritm är den starkaste krypteringsstandarden där möjliga kombinationer för nyckeln som pekar på värdet är 1.1×10^{77}.[3]

3. Milstolpar

Tabellen (tabell 1) nedan är till för att ha utgångspunkter till när projektet ska utföras. Det blir då lättare att arbeta kontinuerligt och alltid ha en plan för vad som implementeras härnäst.

Tabell 1. Lista på milstolpar.

#	Beskrivning
1	Vi kan kryptera strängar och spara i en variabel.
2	Vi kan dekryptera en sträng och visa det i någon mån.
3	Vi kan spara krypterade strängar (användarnamn och lösenord), antingen i en databas eller en fil.
4	Vi kan lägga till nya inloggningsuppgifter i vår lösenordshanterare. Vi kan även visa inloggningsuppgifter.
5	Vi kan ta bort inloggningsuppgifter ur filen/databasen.
6	Vi kan redigera inloggningsuppgifter och spara dom på nytt.
7	Vi har ett användargränssnitt som låter användaren göra ovanstående med hjälp av Java Swing. Vi har knappar för de olika funktionerna.
8	Vi har en navigerbar lista som innehåller alla inloggningsuppgifter.
9	Trycker vi på en av inloggningarna så får man mer information om den där bland annat lösenordet framgår. Vi tar nu bort eller redigerar i detta fönster som poppar upp.
10	Användaren kan logga in med huvudlösenordet till lösenordhanteraren och kan sedan se sina spara lösenord. Knapp för utloggning finns även. (Här har vi kärnprogrammet.)
11	Nu är lösenordet dolt och man behöver trycka på rutan med prickarna för att få fram lösenordet.
12	När man är inloggad kan man byta huvudlösenordet.

[3]Atpinc. *How does AES-256 encryption work to protect your data.* How does AES-256 encryption work to protect your data (atpinc.com), 21-04-29.

13	I detta steg lägger vi mer tid på inloggningsfönstrets utseende. T.ex. logo, snygg text, osv.
14	Kopieringsknapp som kopierar det valda lösenordet.
15	Man kan välja olika kategorier för olika typer av inloggningsuppgifter.
16	Vi kan generera ett lösenord åt användaren där användaren får välja längd och antal av olika symboler.
17	Användaren kan favorisera olika lösenord.
18	Programmet visar lösenord som är identiska och kan föreslå att man ändrar dom.
19	Programmet kan kolla hur bra lösenordet är enligt olika standarder.
20	Sökfunktion på olika sorters inloggningsuppgifter.
21	Det finns en back-up funktion för att kopiera krypterade inloggningsuppgifter till en back-up-fil.
22	Användaren kan importera back-up-filen för att dekryptera inloggningsuppgifterna.
23	Inaktivitet i en viss tid kan programmet låsa sig och inloggningsfönstret visas igen.
…	

4. Övriga implementationsförberedelser

Här förklaras hur programmet ska utformas och fungera. Först kommer ett inloggningsfönster upp som frågar om ett huvudlösenord. Om lösenordet är korrekt ändras fönstret där dekrypterade inloggningsuppgifter kommer upp i en skrollbar panel. Här kan man trycka på olika knappar som har olika funktioner, så som att ta bort, lägga till eller redigera olika inloggningsuppgifter som finns i panelen. Man kan även trycka på en inloggning där mer information om den dyker upp. Det finns en knapp för utloggning där login-fönstret återigen visas.

Vi vill dela upp olika funktionaliteter.

- En del ska hantera kryptering och dekryptering.
- En del ska sköta inloggning till själva programmet så att det är en säker inloggning och hantera fel huvudlösenord.
- En del ska agera som användargränssnitt (Swing), den kommer innehålla alla menyer, alla funktioner som användaren kan komma åt.
- En del är ur säkerhetsaspekt som kollar om information som finns i programmet är krypterat eller dekrypterat. Om vi vill stänga programmet måste denna klass/del kolla om allting är krypterat. Annars kan vi inte stänga eftersom information blir riskfaktor.
- Vi har en del som hanterar fil/databas, t.ex. skriver in nya krypterade lösenord, hämtar dekrypterade lösenord, osv.

5. Utveckling och samarbete

Vi har i denna grupp kommit fram till några punkter som vi vill utgå ifrån när vi tar oss an projektet och sista punkten är vad vi förväntar oss från projektet.

- Vi har samma ambitionsnivå och siktar mot lite högre betyg. Ingen av oss vill sitta med det här projektet efter sommaren/deadline.
- Vi vill jobba under resurstillfällen och tid utöver dessa för att vara i så bra fas som möjligt. Kvällar och helger är okej med båda, men undviks helst.
- Vi planerar detta löpande och sätter upp deadlines efter hur det går. När vi har kommit igång och får en uppfattning av hur lång tid milstolpar tar att utföra kan vi sätta upp deadlines lite längre fram.
- Personliga skäl är godtagbart för att inte kunna arbeta på projektet.
- Vi kommer att jobba tillsammans, men kommer att skriva på olika delar. Oftast kommer vi prata och ta hjälp av varandra under tiden vi jobbar med projektet. Några enstaka gånger kanske man jobbar helt själv. Vi båda är relativt nya till denna sorts programmering och har ingen preferens för vilken del man ska jobba med.
- Vi tror att projektet kommer bli roligt, men svårt.

Projektrapport

6. Implementationsbeskrivning

I det här avsnittet beskrivs det i vilken grad varje milstolpe har utförts på skala helt, delvis eller inte alls. I tabellen (tabell 2) nedan finns beskrivningen för varje milstolpe och i vilken grad den här utförd.

6.1. Milstolpar

I det här avsnittet beskrivs det i vilken grad varje milstolpe har utförts på skala helt, delvis eller inte alls. I tabellen (tabell 2) nedan finns beskrivningen för varje milstolpe och i vilken grad den här utförd.

Tabell 2. Tabellen visar graden av genomförande för alla milstolpar.

Milstolpe	**Grad av genomförande**
1	*Vi kan kryptera strängar och spara i en variabel.* Milstolpen är helt genomförd. Nu sparar vi dock den krypterade strängen (lösenordet) i ett fält i klassen *Account*.
2	*Vi kan dekryptera en sträng och visa det i någon mån.* Milstolpen är helt genomförd. Nu finns en klass *Decrypter* som har hand om all dekryptering av strängar.
3	*Vi kan spara krypterade strängar (användarnamn och lösenord), antingen i en databas eller en fil.* Milstolpen är helt genomförd. Alla konton sparas på en json-fil som sedan kan läsas in när programmet startas. (**Förklara vad en json-fil är??**)
4	*Vi kan lägga till nya inloggningsuppgifter i vår lösenordshanterare. Vi kan även visa inloggningsuppgifter.* Milstolpen är helt genomförd.
5	*Vi kan ta bort inloggningsuppgifter ur filen/databasen.* Milstolpen är helt genomförd.
6	*Vi kan redigera inloggningsuppgifter och spara dom på nytt.* Milstolpen är helt genomförd.
7	*Vi har ett användargränssnitt som låter användaren göra ovanstående med hjälp av Java Swing. Vi har knappar för de olika funktionerna.* Milstolpen är helt genomförd. Vi kan utveckla användargränssnittet, men det gör vi vid senare milstolpe.
8	*Vi har en navigerbar lista som innehåller alla inloggningsuppgifter.* Milstolpen är helt genomförd. Det finns en skrollbar lista där alla konton finns.

9	*Trycker vi på en av inloggningarna så får man mer information om den där bland annat lösenordet framgår. Vi tar nu bort eller redigerar i detta fönster som poppar upp.* Milstolpen är till viss del genomförd. Om vi trycker på ett konto kommer information upp på rutan bredvid. Alltså finns inget fönster som poppar upp och vi tar bort och redigerar fortfarande med knapparna. Programmet håller koll på vilket konto som är i fokus när man trycker på knapparna.
10	*Användaren kan logga in med huvudlösenordet till lösenordhanteraren och kan sedan se sina spara lösenord. Knapp för utloggning finns även. (Här har vi kärnprogrammet.)* Milstolpen är helt genomförd. Man kan logga in till huvudprogrammet. Däremot finns ingen knapp för utloggning då vi nu stänger programmet med krysset på fönstret.
11	*Nu är lösenordet dolt och man behöver trycka på rutan med prickarna för att få fram lösenordet.* Milstolpen är inte genomförd.
12	*När man är inloggad kan man byta huvudlösenordet.* Milstolpen är inte genomförd.
13	*I detta steg lägger vi mer tid på inloggningsfönstrets utseende. T.ex. logo, snygg text, osv.* Milstolpen är inte genomförd.
14	*Kopieringsknapp som kopierar det valda lösenordet.* Milstolpen är inte genomförd.
15	*Man kan välja olika kategorier för olika typer av inloggningsuppgifter.* Milstolpen är inte genomförd.
16	*Vi kan generera ett lösenord åt användaren där användaren får välja längd och antal av olika symboler.* Milstolpen är inte genomförd.
17	*Användaren kan favorisera olika lösenord.* Milstolpen är inte genomförd.
18	*Programmet visar lösenord som är identiska och kan föreslå att man ändrar dom.* Milstolpen är inte genomförd.
19	*Programmet kan kolla hur bra lösenordet är enligt olika standarder.* Milstolpen är inte genomförd.
20	*Sökfunktion på olika sorters inloggningsuppgifter.* Milstolpen är inte genomförd.
21	*Det finns en back-up funktion för att kopiera krypterade inloggningsuppgifter till en back-up-fil.* Milstolpen är inte genomförd.
22	*Användaren kan importera back-up-filen för att dekryptera inloggningsuppgifterna.* Milstolpen är inte genomförd.
23	*Inaktivitet i en viss tid kan programmet låsa sig och inloggningsfönstret visas igen.* Milstolpen är inte genomförd.

...

Tabell 3. Tabellen visar graden av genomförande för alla milstolpar.

6.2. Dokumentation för programstruktur, med UML-diagram

I det här avsnittet går vi in i funktionaliteten på ett mer grundligt sätt. Vi har innan förklarat ungefär hur vi vill att programmet ska fungera och vad det ska göra. Nu är det dags att förklara hur exakt vi byggt upp programmet i termer om java och programmering. Vi vill visa hur klasserna i programmet arbetar tillsammans och vilken roll de spelar i förhållande till varandra. Avsnittet är uppdelat i två rubriker där *Programstruktur* ger en övergripande programstruktur för ett få en tydligare bild av hur programmet är uppbyggt. Det andra avsnittet går djupare in på programstrukturen och förtydligar även klassernas förhållande till varandra.

6.2.1. Programstruktur

Programmet är uppbyggt på det sättet att klasser har hand om olika delar och funktionalitet. Programmet är huvudsakligen uppdelat i ett visuellt plan och ett logiskt plan. De olika planen vill man gärna inte blanda ihop just för att användaren inte ska behöva veta om det logiska planet. Det logiska planet finns för att sköta kryptografin, spara konton till filer, manipulera listor med konton och många andra saker. Det visuella planet finns för att användaren av programmet ska kunna använda funktionaliteten på lättast möjliga sätt och inte behöva tänka på vad just programmet och koden gör exakt när man till exempel lägger till ett konto. Användaren trycker ju bara på en knapp för att det ska hända något, men det finns mycket annat som händer i just koden.

6.2.1.1 Visuella planet

Det visuella planet består huvudsakligen av en klass *PasswordManagerWindow* som håller koll på vad användaren gör. Klassen har ansvar för ett ganska stort område vilket är en anledning till att klassen innehåller mycket kod. Tillsammans med klassen *WindowManager* bestämmer de sig för vilket fönster som ska visas: programfönstret, inloggningsfönstret eller uppstartsfönstret. *WindowManager* kontrollerar om det finns en fil som har ett hashat lösenord i sig. Om det inte är fallet, skickar *WindowManager* med information till *PasswordManagerWindow* om att ett uppstartsfönster ska visas. Annars startar klassen ett inloggningsfönster för att verifiera att rätt användare försöker ta sig in i lösenordshanteraren. Själva autentiseringen av lösenordet hanteras av det logiska planet där informationen om vilket lösenord som angivits skickas med. Redan här kan vi se den tydliga uppdelningen av uppgifter i olika klasser. Ansvaret som *PasswordManagerWindow* bär är endast att reagera på användaren aktioner och visa information som är viktigt för användaren.

Klassen *PasswordManagerWindow* är en viktig del av programmets utformning. Klassen kan registrera knapptryck från användaren, men även visa information från det logiska planet som är viktigt att veta för användaren som nämnts tidigare. Det viktiga här är att den här klassen inte ska ha tillgång till för mycket information från det logiska planet, det kan vara känslig information till exempel om hur och var lösenorden sparas, vilket kan utnyttjas om det hamnar i fel händer. Av just den anledningen finns det en klass, *LogicHandler*, som fungerar som en mellanhand till det logiska planet och det visuella planet. *LogicHandler* har tillgång till information från det logiska planet och kan på så sätt distribuera den information som *PasswordManagerViewer* behöver från det logiska planet. Mellanhanden innehåller pekare till alla logiska delar och kan på det sättet skicka information från det visuella planet, till

exempel när användaren trycker på en knapp, till den logiska delen som ska ta hand om den specifika typen av information.

6.2.1.2 Logiska planet

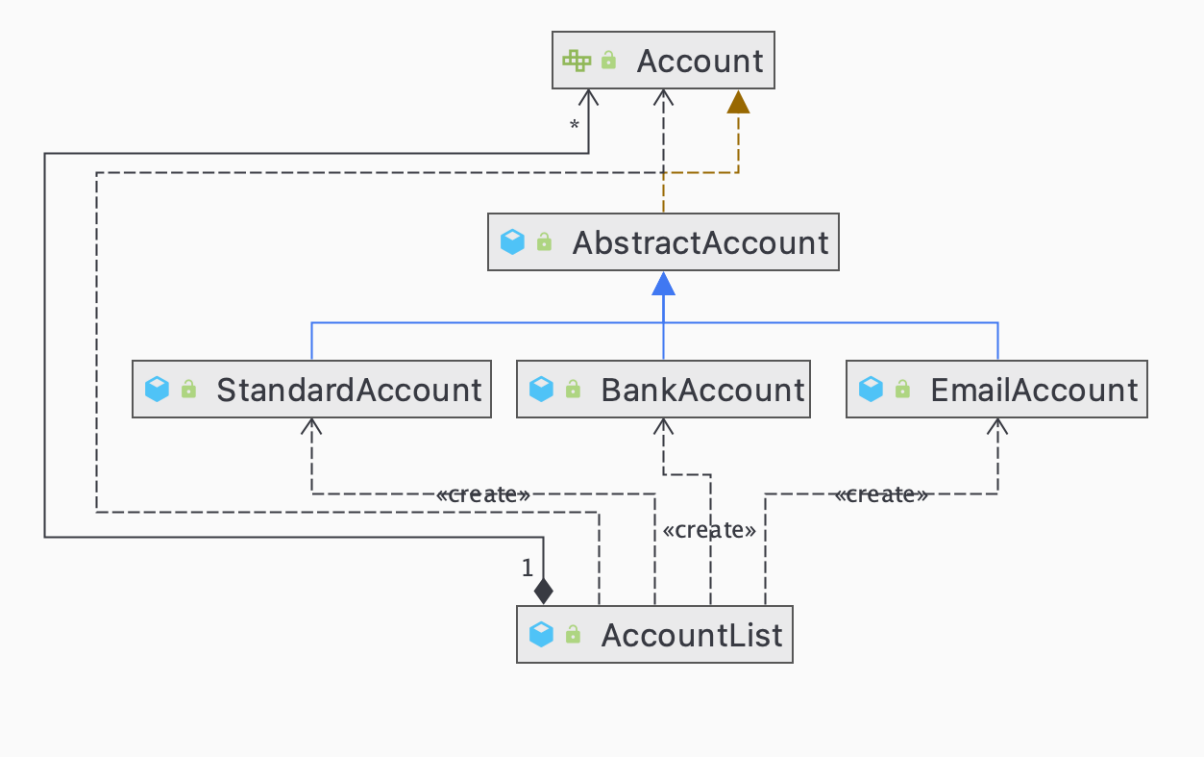
Det logiska planet är uppdelat i olika delar. Här finns flera klasser som tar hand om olika delar av programmets logiska funktioner. Generellt är uppdelningen följande: klasser som tar hand om kryptografin, klasser som sparar och manipulerar användarens olika konton och klasser som har hand om inloggningen. Kryptografiklasserna har uppgifterna att skapa nycklar till kryptering och dekryptering för att kunna kryptera och dekryptera strängar. De här klasserna innehåller även information om krypteringsalgoritmen som används under processen. Det är därför viktigt att hålla den informationen privat endast för dessa klasser. Annars finns det risk att den informationen kan användas för att dekryptera någon annans lösenord. Den andra delen av klasser har uppgifterna att skapa konton som objekt, ta bort eller redigera existerande konton och spara konton i både en lista och på fil.

För att få en bild av hur det fungerar och hur det visuella planet och det logiska planet arbetar tillsammans är ett exempel på när programmet används i sin ordning. För att lägga till ett konto i sin lösenordshanterare trycker användaren på knappen *Add account*. Klassen *PasswordManagerViewer* registrerar knapptrycket och frågar användaren om användarnamn och lösenord. Informationen om att en knapp har tryckts ned skickas sedan till mellanhanden, *LogicHandler*, som utreder vad för sorts information som inkommit. Mellanhanden får klart för sig att användaren vill lägga till ett konto och skickar information vidare till klassen *AccountList*. Den här klassen skickar vidare informationen om vad användaren vill ha för användarnamn och lösenord till en av kontotypsklasserna, beroende på vilken användaren valt att lägga till, för att skapa ett konto-objekt. När kontot är skapat går programmet tillbaka till *AccountList* för att lägga till kontot i listan och spara den på filen. Nu är allt i det logiska planet utfört och programmet går tillbaka till det visuella planet för att visa det nya kontot i listan.

6.2.2. Relaterade klasser

Här beskrivs alla relaterade klasser i detalj. Samarbeten mellan klasser, hierarkier och subtypspolymorfism är några saker som diskuteras i avsnittet. Figurerna i avsnittet är avsett för att enklare förstå vad som förklaras i texten.

6.2.2.1 Kontohantering



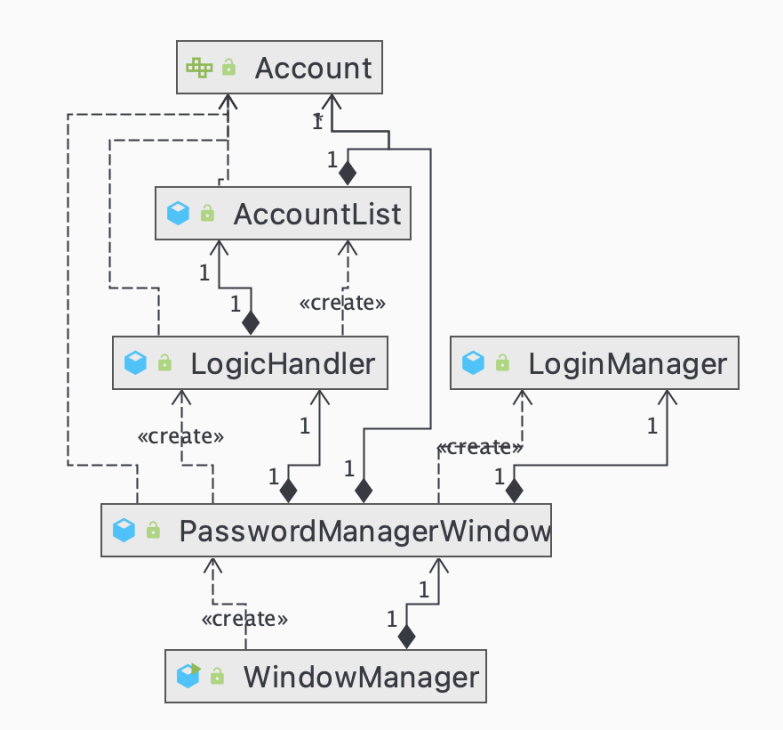
Figur 2. Bilden är ett UML-diagram som visar kontostrukturen och alla dess sammanhängande klasser i det logiska planet.

En vital del i en lösenordshanterare är att kunna hantera alla konton som användaren väljer att lägga till, redigera eller ta bort. Figur 2 ska illustrera en viktig del för att användaren ska kunna göra just det, själva hanteringen av att skapa konto-objekt och manipulera listan med dessa objekt. *Account* är ett gränssnitt som gör det möjligt för oss att beskriva flera olika konton utan att specificera kontotypen. *AbstractAccount* implementerar alla metoder som *Account* innehåller. *AbstractAccount*s främsta uppgift är att agera som en bas för alla olika kontotyper: *StandardAccount, BankAccount* och *EmailAccount*. Klassen innehåller dels metoder för att skapa ett konto-objekt, dels för att manipulera ett konto-objekt genom att till exempel ändra användarnamnet eller lösenordet. Vad detta gör är att dessa specifika kontotypsklasser kan alla använda en och samma kod för att skapa sig själva medan vi samtidigt kan hantera sådana här olika kontotyper på det sätt vi vill. Meningen med att dela upp det i olika kontotyper som ärver från en superklass är att det finns likheter men även olikheter. Likheterna är att alla byggs upp på samma sätt med ett användarnamn av något slag och ett lösenord. På det sättet sparar vi repeterande kod. Olikheterna är att sådana här olika kontotyper innehåller har distinkta skillnader i vad det är för sorts information som ett visst konto ska innehålla. Vi vill kunna göra det möjligt för användaren att spara olika sorters kontotyper som har dessa skillnader och det gör vi bäst med den här sortens subtypspolymorfism. De specifika skillnaderna i det här fallet är att *EmailAccount* tar in en mailadress och en domän tillsammans med användarnamnet och lösenordet. Vidare har *BankAccount* ett fält som ska beskriva kontonumret för bankkontot. Eftersom alla kontotyper har samma bas kan vi därmed använda oss av subtypspolymorfism och utöka klasserna som är i behov av det.

Klassen *AccountList* är ansvarig för att spara konton som skapas i array-lista som finns som ett privat fält i klassen. Array-listan innehåller alltså *Account*s och det är därför det är viktigt att objekten implementerar gränssnittet. Det här gör det möjligt för listan att inte behöva veta alls vilket typ av konto-objekt som läggs till, utan bara att det är *något* typ av konto.

AccountList har även i uppgift att kunna skriva alla konton på fil så att användaren kan stänga av programmet och kunna se sina tillagda konton igen. Det gör klassen med en metod som heter *saveOnFile* som använder sig av *Gson* för att spara vår data om alla konton på en json-fil. Ett problem med *Gson* är att om man vill skriva till en fil med olika kontotyper, har inte *Gson* verktyg för att verifiera att dessa konton verkligen är konto-objekt och kan därför inte läggas till i listan igen utan att kasta ett undantag. Av den anledningen finns en klass som heter *AccountAdapter* som i stort sett verifierar att det är ett konto-objekt vid skrivning till fil och läsning av fil. Vid läsning av fil skapar klassen konton utifrån data i filen för att sedan lägga in kontona i listan. Av just den anledningen innehåller *AbstractAccount* två konstruktorer. Att skapa ett helt nytt objekt behöver annan information än om man skapar ett objekt utifrån data från objekt som redan har skapats, även om båda kan dela viss information. Utöver *AccountLists* uppgifter som redan beskrivits har klassen även tillgång till att ändra klassens användarnamn och lösenord med hjälp av *AbstractAccounts* metoder, ta bort konton helt från listan och kan hämta ut konton från listan vid behov.

6.2.2.2 Samverkan mellan visuella och logiska lagret



Figur 3. Bilden är ett UML-diagram som visar det visuella planet och alla dess sammanhängande klasser. Diagrammet visar även hur det visuella planet hänger samman med mellanhanden till det logiska lagret.

Uppdelningen av klasserna som visas ovan är alla klasser som hänger samman med det visuella planet, där *PasswordManagerWindow* är själva kärnan. För att få en bild av hur dessa klasser hör tillsammans kan man titta på figur 3 ovan.

Klassen *WindowManager* ansvarar för att rätt fönster visas för användaren vid rätt tidpunkt. Om det till exempel är första gången som lösenordshanteraren startas ska ett välkomstfönster visas och om det inte är första gången ska ett inloggningsfönster visas. Det är även *WindowManagers main* metod som körs för att lösenordshanteraren ska startas.

I *WindowManagers* konstruktor skapas fönsterobjektet *PasswordManagerWindow*. Metoden *initManager* ansvarar för att rätt fönster öppnas vid rätt tillfälle. Det metoden gör är att den kollar med hjälp av den andra metoden *isFirstTimeStartup* om det finns ett sparat

lösenord i en viss fil. Om det finns ett lösenord ska inloggningsfönstret startas och om det inte finns ett tidigare sparat lösenord ska välkomstfönstret startas. Detta görs genom att antingen metoden *doFirstTimeStartup* eller metoden *startLoginWindow* anropas. *DoFirstTimeStartup* öppnar sedan fönstret genom att kalla på *PasswordManagerWindows show*-metod och skicka in en enum för vilken fönstertyp som ska öppnas, i det här fallet *SETUP* fönstret. Metoden *startLoginWindow* fungerar på samma sätt och kallar också på *PasswordManagerWindows show*-metod och skickar in enumen *LOGIN* för att visa inloggningsfönstret i stället.

Klassen PasswordManagerWindow är programmets fönsterobjekt och hanterar hela det grafiska användargränssnittet. Det är här den faktiska lösenordshanteringen sker. Efter att *PasswordManagerWindow* har öppnats behöver den inte längre kommunicera med *WindowManager* och den sköter kommunikation med det logiska lagret på egen hand. *PasswordManagerWindow* skapar objektet *LogicHandler* som agerar som en mellanhand mellan det visuella och det logiska lagret och informationsdelningen sker via den. Den tidigare nämnda metoden show i klassen tar in enumsens *LOGIN*, *SETUP* eller *PASSWORD_MANAGER* och motsvarande vilken fönstertyp den fick in initialiserar den rätt saker för den typen av fönster. Om det är ett inloggningsfönster som ska öppnas skapar *PasswordManagerWindow* mellanhandsobjektet *LoginManager*. När användaren skriver in lösenordet i inloggningsfönstret och trycker på knappen login skickas lösenordet vidare till *LoginManager via klassens authenticateLogin* metod. Om lösenordet är korrekt returnerar *LoginManager* true och huvudfönstret för lösenordshanteraren öppnas. Om det är fel lösenord returneras false och användaren får ett meddelande som säger att lösenordet är fel. Vidare när det är ett välkomstfönster som ska öppnas initieras återigen rätt saker för den fönstertypen. Det användaren får göra i välkomstfönstret är att bestämma ett huvudlösenord som sedan ska användas för att logga in och kryptera alla lösenord. När användaren sedan trycker på continue-knappen skapas ett *LogicHandler* objekt som tar in lösenordet i sin konstruktor och sedan sparar ner lösenordet till en fil. Efter det byts fönstret till huvudfönstret.

I huvudfönstret sker den egentliga lösenordshanteringen där användaren kan se alla sina sparade konton och tillhörande lösenord. Användaren kan även spara nya konton, ändra och ta bort konton. *PasswordManagerWindow* kommunicerar med den logiska mellanhanden *LogicHandler* för att utföra alla dessa operationer på listan med konton. I *PasswordManagerWindow* är det metoden *doAction* som gör alla operationer på listan genom *LogicHandler* objektet. Alla knappalternativ har sina egna metoder, och det som alla knappmetoder gör är att de samlar in information som användarnamn och lösenord och skickar sedan det till *doAction*. Metoden *doAction* skickar vidare den informationen till *LogicHandler* via *LogicHandlers doAccountAction* metod och beroende på vilken information som *LogicHandler* får in så utförs de olika operationerna på kontolistan. Efter att kontolistan ändrats uppdaterar *doAction* kontolistan som visas i fönstret genom att hämta den från *LogicHandler* via *getAccounts* metoden.

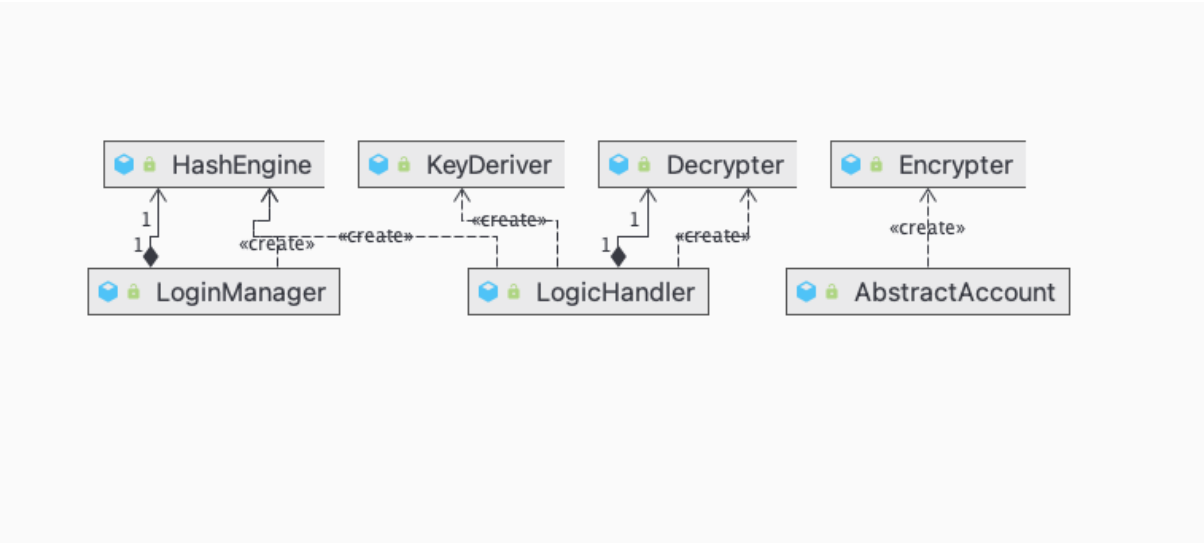
Klassen *LoginManager* är ansvarig för att jämföra användarens sparade lösenord mot det som skrivs in. Den läser in det sparade hashade lösenordet från en fil och sedan hashas det lösenord som användaren skriver in. Efter det går det att jämföra hasherna på de två lösenorden, och om dem matchar är lösenordet korrekt. Lösenordshashningen beskrivs i mer detalj senare. Vid korrekt lösenord returnerar metoden *authenticateLogin* true och vid fel lösenord returnerar metoden false. Det är baserat på detta resultat som *PasswordManagerWindow* sedan agerar.

Klassen *LogicHandler* är mycket viktig för att det visuella lagret ska kunna ändra i kontolistan och hantera konton. I *LogicHandlers* konstruktor skapas objektet *AccountList* som innehåller själva kontolistan. Objektet läses in från fil med gson och om det inte finns någon fil skapas ett nytt *AccountList* objekt med tom kontolista. Det är med metoderna som finns i *AccountList* som *LogicHandler* kan skapa konton, ändra konton eller ta bort konton. *LogicHandlers* metod *DoAccountAction* tar in nödvändig information som till exempel ett nytt användarnamn och sedan baserat på vilken knapp som den också tar in kallar den på rätt metod från *AccountList* klassen. Om ett nytt konto ska skapas körs *AccountList* metoden *addAccount*, när ett konto ska tas bort körs *removeAccount* och när ett konto ska ändras på körs *editAccount*. *Logichandler* har också metoden *readJsonAccountList* som läser in *AccountList*-objektet från fil när programmet startar. Vid första uppstart av programmet får användaren välja ett huvudlösenord till lösenordshanteraren och det är metoden *saveHashtoFile* i *LogicHandler* som ser till att användarens nya lösenord hashas och sedan sparas ner till fil. Klassen har även getters för hela kontolistan *AccountList* och en för kontolösenord. Metoden *getAccounts* används av *PasswordManagerWindow* för att läsa in kontolistan till en Jlist som sedan visas i fönstret. *PasswordManagerWindow* använder också *LogicHandlers* metod *getAccountPassword* som dekrypterar ett specifikt användarkontos lösenord och returnerar det.

Klassen *AccountList* innehåller en arraylista med alla användarens sparade konton och ser även till att den listan sparas ner till fil. Den har flera metoder som gör det möjligt för *LogicHandler* att ändra i kontolistan och som gör det möjligt att till exempel skapa nya konton. Metoderna *removeAccount* och *getEncryptedAccount* använder arraylistans inbyggda metoder för att ta bort ett konto från listan och för att hämta ett konto på ett specifikt index. Metoderna *addAccount* och *editAccount* lägger till och ändrar i redan existerande konton, detta möjliggörs delvis genom att konto-klasserna har egna metoder för att byta lösenord och ändra användarnamn i konto-objekten. När användaren vill ändra ett lösenord skickas det nya lösenordet först från *PasswordManagerWindow* till *LogicHandler* som skickar vidare det till *AccountLists editAccount* metod som i sin tur skickar lösenordet vidare ytterligare ett steg till konto-klassens *editPassword* metod. Därefter sparas ändringen till filen och kontolistan uppdateras.

Den abstrakta klassen *AbstractAccount* är själva användarkontot och kan bland annat innehålla ett användarnamn och lösenord, samt vilken kontotyp kontot är av. *AbstractAccount* har metoderna *editPassword* och *editUsername* som sätter ett nytt lösenord och ett nytt användarnamn för kontoobjektet. Metoden *editUsername* gör inget mer än att sätta det nya användarnamnet som användarnamnet för kontot. Metoden *editPassword* är något mer avancerad än *editUsername* eftersom den krypterar det nya lösenordet innan det läggs in. Krypteringen kommer förklaras i mer detalj senare i texten. Klassen har även getters för bland annat användarnamnet, det krypterade lösenordet och kontotypen, något som används av *LogicHandler*.

6.2.2.3 Kryptering och hashning och dess roll i det logiska planet



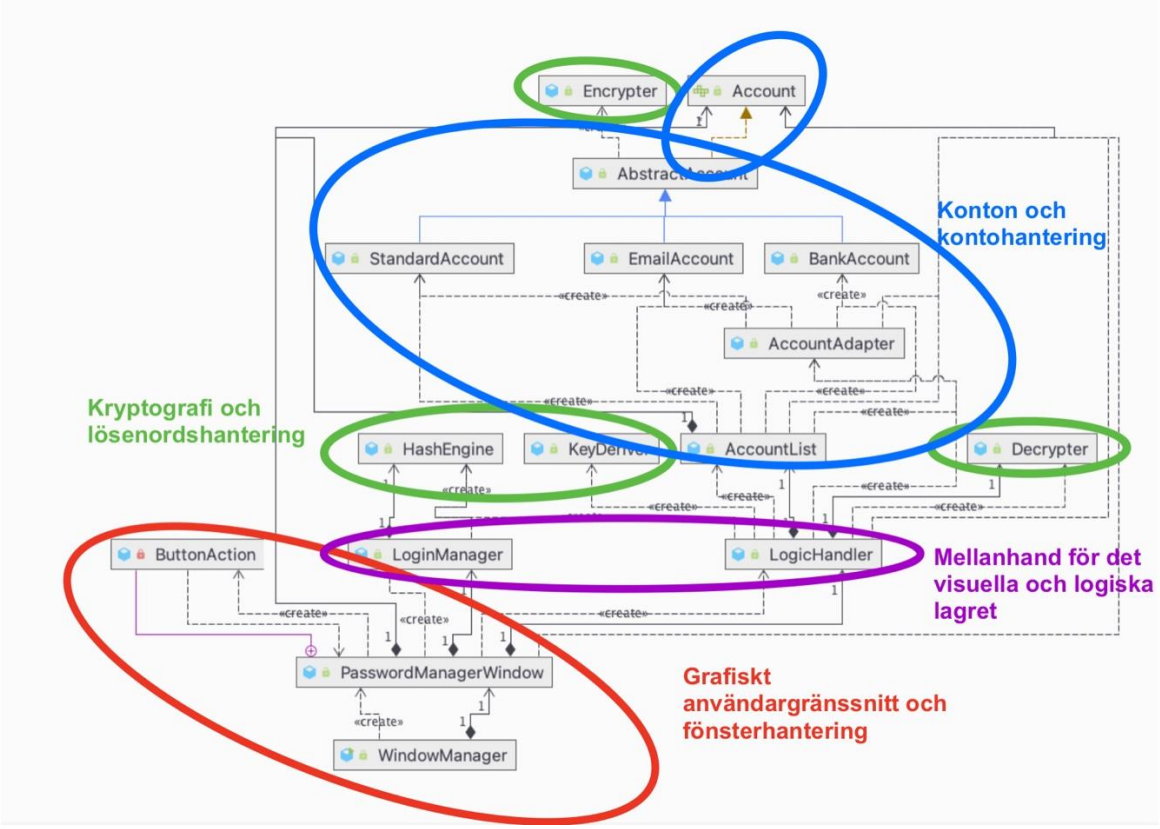
Figur 4. Bilden är ett UML-diagram som visar vilken roll kryptografin spelar i programmets utformning.

För att säkra viktig information från konton behövs några saker. Man måste kunna skydda konton genom att säkerställa att endast en person har tillgång till dessa, i vårt fall sker det i formen av en inloggning. Det måste också finnas något som gör att inte fel person kan se informationen. Det är här kryptografin kommer in. Figur 4 ska illustrera hur olika delar av programmet använder sig av kryptografin. För att säkerställa att viktig information från ett konto inte ska kunna läckas så arbetar *AbstractAccount* nära med klassen *Encrypter*. *Encrypter* har i uppgift att kryptera lösenord till en byte array. Klassen använder sig av en så kallad chiffer, en krypteringsalgoritm (AES-256) och en nyckel vilket gör det möjligt för klassen att returnera ett krypterat lösenord i metoden *encryptPassword*. Nyckeln är nödvändig då om ett lösenord ska dekrypteras utan samma nyckel som angavs vid kryptering finns det ingen möjlighet att komma åt lösenordet. Nyckeln skriver om någon sorts information till chiffer-text vid kryptering och det omvända vid dekryptering. Metoden returnerar även en initialiseringsvektor i syfte av att göra vårt krypterade lösenord ännu mer unikt. *AbstractAccount* kan då skapa ett *Encrypter*-objekt och lösenordet som användaren har skrivit in för att skapa sitt konto krypteras med hjälp av *Encrypter* och sparas i ett privat fält i *AbstractAccount*. Här sparas även initialiseringsvektorn som ett privat fält. Detta tillåter oss att spara konton på fil utan att behöva oroa oss för att lösenordet ska avslöjas då det redan har krypterats. Därmed är det användaren som väljer när denne vill se ett lösenord.

Ett musklick på ett av kontona i listan som visas framför användaren gör att en metod i *LogicHandler* anropas. Metoden heter *getAccountPassword* och ligger i klassen som agerar mellanhand för att vi inte vill att det visuella lagret ska ha direkt tillgång till något som är en del av det logiska lagret. Metoden returnerar alltså ett läsbart lösenord som *PasswordMangerViewer* tar hand om så att användaren kan ta del av informationen. Det här utför metoden genom att använda pekaren till *Decrypter*-klassen och använda dess metod för att dekryptera ett givet och krypterat lösenord. För att ens kunna dekryptera behöver klassen ha tillgång till ett chiffer, exakt samma krypteringsalgoritm och en nyckel. Hur får vi tag på nyckeln som vi använde för att kryptera ett lösenord? Jo, nyckeln erhålls från huvudlösenordet som användaren valt. På det sättet kan vi alltid erhålla en nyckel utan att behöva spara den och på det sättet skydda den mot säkerhetshot. Det här utförs av klassen *KeyDeriver* som innehåller en metod *deriveKey* som har till uppgift att ta in ett läsbart lösenord och ett givet salt för att sedan med hjälp av en hashnings-algoritm (PBKDF2WithHmacSHA1) och en

krypteringsalgoritm (AES-256) returnera en krypteringsnyckel. Med krypteringsnyckeln i behåll kan vi dekryptera något sparat lösenord från ett existerande konto och dess initialiseringsvektor. Metoden returnerar ett lösenord i form av en byte-array och för att användaren ska kunna se lösenordet i läsbar text görs det till en *String* i *LogicHandler*.

För att kunna ha någon sorts inloggning till programmet måste ett lösenord sparas någonstans för att sedan jämföra med lösenordet som användaren skriver in. Det är här klassen *HashEngine* kommer in. Från första start av programmet väljer användaren ett icke-tomt lösenord. När användaren väljer ett lösenord skickas lösenordet till *LogicHanddler*. Metoden *saveHashToFile* tar in det läsbara lösenordet och använder Gson, som nämnts tidigare, och *HashEngine* för att skriva lösenordet till en fil. Vi har valt att hasha lösenordet och sedan spara det till filen för att ett hashat lösenord är omöjligt att reversera vilket gör det säkert att spara det till filen utan hot. Hashningen sker som sagt i *HashEngine* och i metoden *generateHash* som använder sig av ett givet salt och en hashnings-algoritm för att sedan returnera en byte-array. Det hashade lösenordet sparas tillsammans med saltet som användes vid hasningen och ett deriveringssalt som används vid deriveringen av krypteringsnyckeln. Dessa salt läses sedan in nästa gång programmet startas och vi har nu givna salt som kan användas varje gång en krypteringsnyckel behöver erhållas eller när vi vill hasha ett givet lösenord. Nu när vi har ett huvudlösenord som kan användas vid inloggning måste vi kunna jämföra det med något som användaren skriver in. Programmet kontrollerar ett givet lösenord genom att skicka att *PasswordManagerViewer* skickar informationen till en klass som kallas *LoginManager* som sköter allt med inloggning att göra. *LoginManager* har en metod som har hand om autentiseringen av lösenordet. Det första metoden gör är att hasha det testade lösenordet från användaren och med saltet som vi också hade sparat. Sedan läses vårt sparade hashade lösenord in från filen med metoden *readHashPasswordFile*. Två lika lösenord ska ha exakt samma hashning och det gör att vi kan jämföra dessa olika hashade strängar och metoden returnerar ett sanningsvärde beroende på om hashningarna stämmer överens. Informationen skickas tillbaka till *PasswordManagerViewer* och bestämmer om vi kan fortsätta till huvudprogrammet eller inte.



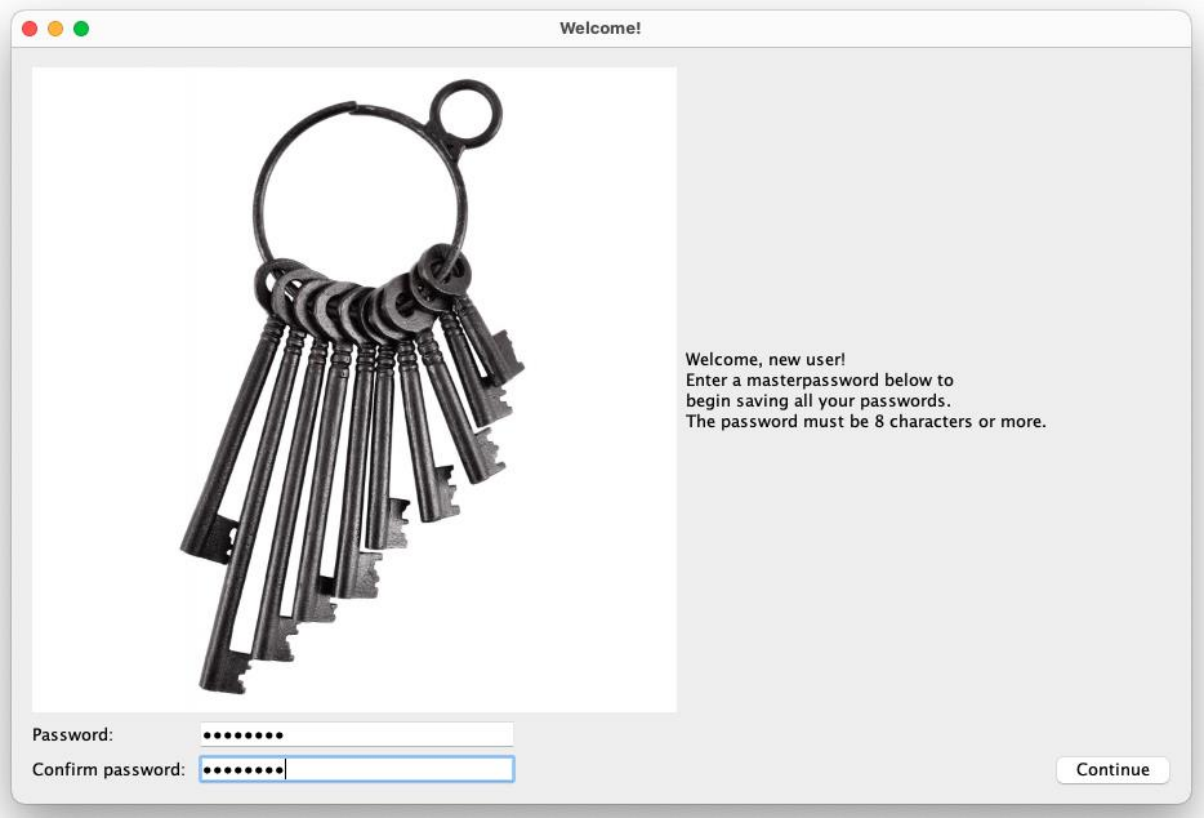
Figur 5. Bilden är ett UML-diagram som visar hela programmets översikt och alla dess sammanhängande klasser.

Figur 5 ska ge en helhetsbild av lösenordshanterarens struktur och uppbyggnad samt samverkan mellan de olika delarna.

7. Användarmanual

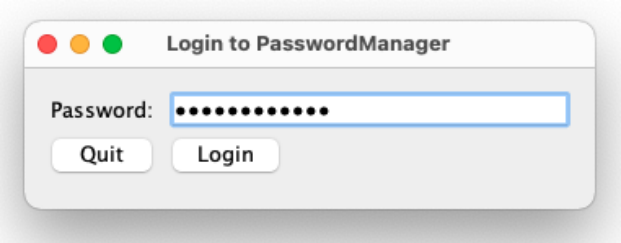
En användarmanual är alltid en bra beskrivning på hur exakt programmet fungerar för användaren. Därför kommer just en beskrivning av hur man använder lösenordshanteraren och vilka funktionaliteter som finns tillgängliga för användaren.

Om det är din första gång du öppnar programmet kommer ett fönster upp (figur 6) som kräver att du skapar ett huvudlösenord som senare kommer att vara din inloggning till lösenordshanteraren. Utan ditt huvudlösenord kommer du inte att komma åt dina framtida sparade konton. I fältet *New masterpassword* skriver du det huvudlösenord du vill ha och i fältet *Confirm password* skriver du in samma lösenord för att vara säker på att du skrivit in det rätta lösenord du vill ha.



Figur 6. Uppstart av lösenordshanteraren.

När du är nöjd med lösenordet klickar du på knappen *Continue* för att få logga in på din lösenordshanterare med huvudlösenordet du alldeles nyss angav. Figur 7 visar hur inloggningsfönstret ser ut.

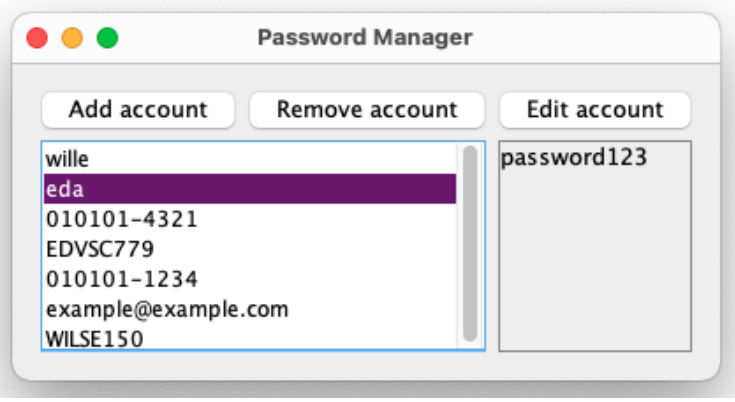


Figur 7. Inloggningsfönstret till lösenordshanteraren.

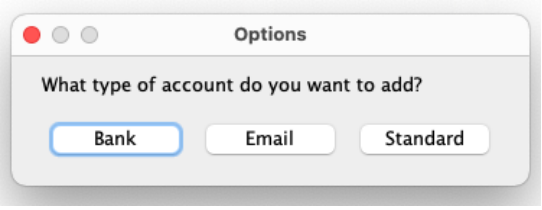
Anger du fel lösenord i fältet *Password* dyker en ruta upp som säger att fel lösenord har angivits. Tryck på knappen *Ok* för att försöka igen. När du väl angivit rätt lösenord trycker du på knappen *Login* för att komma till själva programmet. Här ser du olika knappar som har olika funktioner. Tryck på *Add account* för att lägga till ett konto. Programmet kommer fråga dig om vilken typ av konto du vill lägga till, ett *StandardAccount, EmailAccount* eller *BankAccount,* något som syns i Figur 9. Efter att ha valt kontotyp får du skriva in ett användarnamn och ett lösenord, tryck sedan på *OK* för att skapa kontot. Nu syns det nya kontot i den skrollbara listan åt vänster. Trycker du på kontot i listan kan du se ditt lösenord för det kontot.

Trycker du på knappen *Remove account* tar du bort kontot som är i fokus. Du kan se vilket konto som är i fokus antingen genom att se vilket lösenord som visas i den högra rutan eller genom att observera vilket konto som är skuggad. Om du tar bort ett konto kan du se att kontot försvinner från listan med konton.

Det finns även en knapp *Edit account* som låter dig redigera konton. En ruta dyker upp som ger dig möjligheter att välja att redigera användarnamnet, lösenordet eller båda. Skriv in ditt önskade användarnamn och/eller lösenord för att sedan spara det genom att trycka på *OK*. Figur 8 nedan visar hur huvudprogrammet ser ut.



Figur 8. Lösenordshanterarens (huvudprogrammet) fönster.



*Figur 9. Konto-alternativen som syns efter att knappen **Add account** tryckts.*

För att avsluta programmet trycker du helt enkelt på krysset längst upp åt höger.